

Iterators and Comparators



SoftUni Team
Technical Trainers
Software University
<http://softuni.bg>



Table of Contents

1. Variable Arguments

2. Iterators

- Iterator
- ListIterator

3. Comparators

- Comparable



sli.do

#JavaFundamentals

Variable Arguments (varargs)

- Allows the method to accept **zero** or **multiple** arguments

```
static void display(String... values) {  
    System.out.println("display method invoked");  
}  
  
static void main() {  
    display();  
    display("first");  
    display("multiple", "Strings"); }  

```



Ellipsis syntax

Variable Arguments Rules

- There can be **only one** variable argument **in the method**.
- Variable argument **must** be the **last argument**.

```
static void display(int num, String... values) {  
    System.out.println("display method invoked");  
}
```

```
void method(String... a, int... b) {}//Compile time error
```

```
void method(int... a, String b) {}//Compile time error
```


Problem: Book

- Create a class Book, which have:
 - Title
 - Year
 - Authors
- Use **only one constructor** for book
- Authors can be **anonymous**,
one or **many**

Book
<pre>-title: String -year: int -authors: List<String></pre>
<pre>-setTitle(String) -setAuthors(String...) -setYear(int) +getTitle(): String +getYear(): int +getAuthors(): List<String></pre>

Solution: Book

```
//TODO: Add fields  
public Book(String title, int year, String... authors) {  
    this.setTitle(title);  
    this.setYear(year);  
    this.setAuthors(authors);  
}
```

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/523#0>

Solution: Book(2)

```
//TODO: Add all other getters and setters
private void setAuthors(String... authors) {
    if (authors.length == 0) {
        this.authors = new ArrayList<String>();
    } else {
        this.authors = new ArrayList<>(Arrays.asList(authors));
    }
}
```

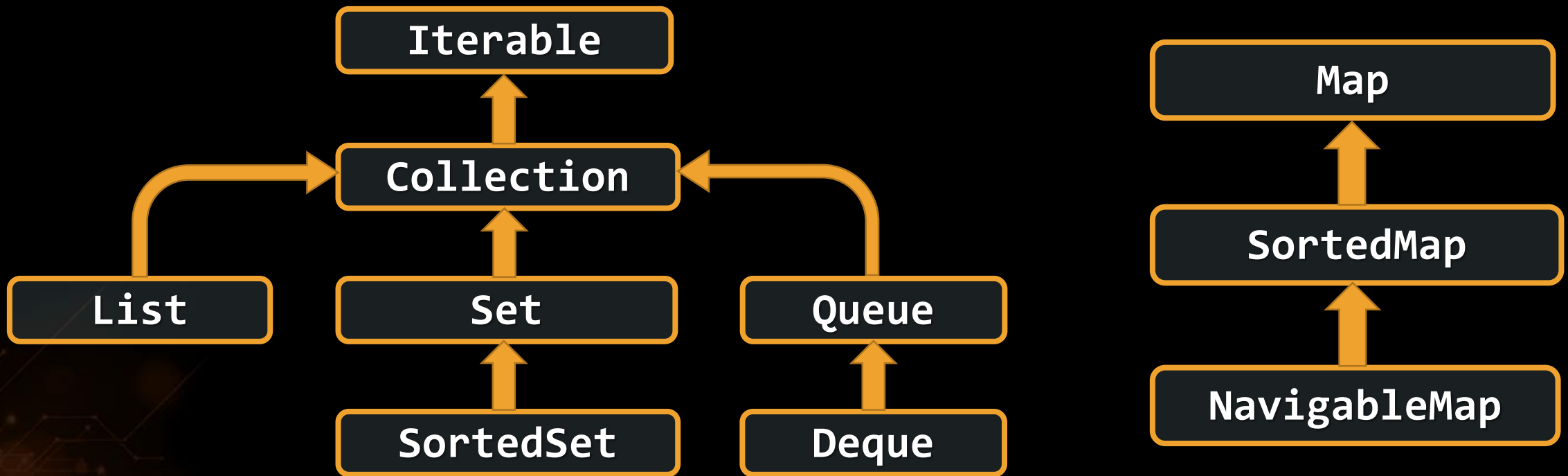
Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/523#0>



Iterable<T> and Iterator<T>

Collections Hierarchy

- **Inheritance** leads to **hierarchies** of classes and/or interfaces in an application:



Iterable<T>

- Root interface of the Java collection classes
- A class that implements the **Iterable<T>** can be used with the new **for loop**

```
List list = new ArrayList();  
for(Object o : list) {  
    //do something o;  
}
```

Iterable<T> Methods

- Abstract methods

- **iterator()**

```
public interface Iterable<T> {  
    public Iterator<T> iterator();  
}
```

- Default methods


- **forEach**(Consumer<? **super** T> action)

- **splitterator()** - used for parallel programming

Iterator<T>

- Enables you to cycle through a collection
- Nested class for **Iterator<T>**

```
public class Library<T> implements Iterable<T> {  
    private final class LibIterator implements Iterator<T> {}  
}
```



- Don't implement both **Iterable<T>** and **Iterator<T>**

```
class MyClass implements Iterable<T>, Iterator<T> {}
```



Problem: Library

- Create a class Library, which implements **Iterable<Book>**
- Create nested class LibIterator, which implements **Iterator<Book>**

```
<<Iterator<Book>>>  
LibIterator
```

```
-counter: int
```

```
+hasNext(): Boolean  
+next(): Book
```

```
<<Iterable<Book>>>  
Library
```

```
-books: Book[]
```

```
+iterator(): Iterator<Book>
```

Solution: Library

```
public class Library<Book> implements Iterable<Book> {  
    private Book[] books;  
    public Library(Book... books) {  
        this.books = books;  
    }  
    public Iterator<Book> iterator() {  
        return new LibIterator();  
    }  
} //TODO: Add nested iterator, look for it on next slide
```

Check your solution here: <https://judge.softuni.bg/Contests/Practice/Index/523#0>

Solution: Library (2)

```
private final class LibIterator implements Iterator<Book> {  
    private int counter = 0;  
    public boolean hasNext() {  
        if(this.counter < books.length) { return true; }  
        return false;  
    }  
}
```

Solution: Library (3)

```
public Book next() {  
    counter++;  
    return books[counter - 1];  
}  
}
```



Iterable<T> and Iterator<T>

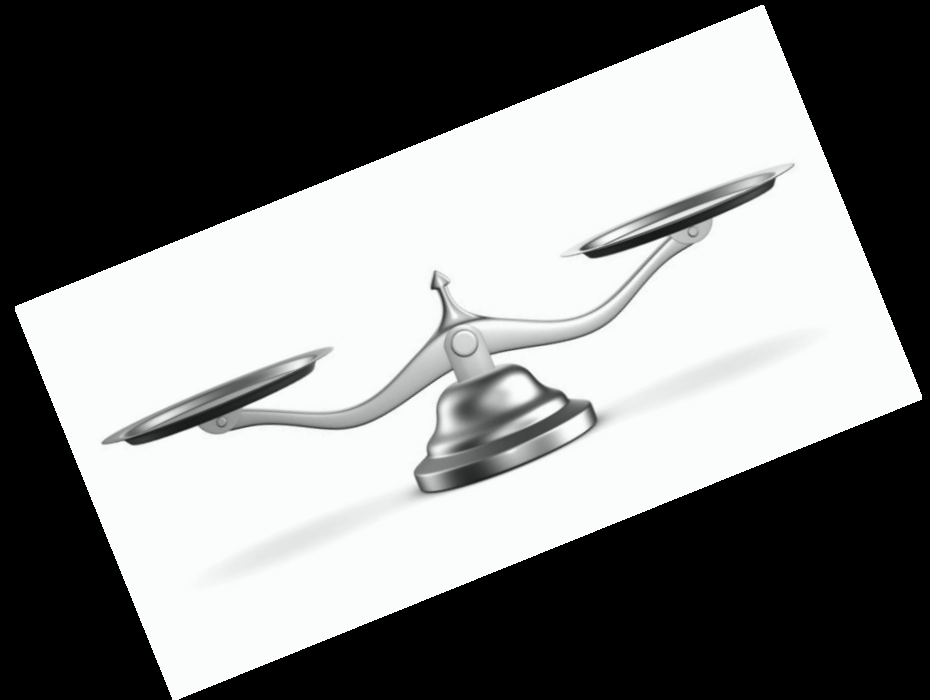
Live Exercises in Class (Lab)



Comparable<T> vs Comparator <T>

Comparator <E> vs Comparable <E>

- Comparator provides a way for you to **provide custom comparison logic** for types that you have no control over
 - **Multiple** sorting sequence
 - **Doesn't affect** the original class
 - **compare()** method



Comparator <E> vs Comparable <E>

- Comparable allows you to specify how objects **that you are implementing** get compared
 - **Single** sorting sequence
 - **Affects** the original class
 - **compareTo()** method



Comparable <E>

- Allows you to specify how objects that **you are implementing** get compared.

```
class Student implements Comparable<Student> {  
    private String name;  
    private int age;  
    public int compareTo(Student st) {  
        if (this.age == st.age) { return 0; }  
        else if (this.age > st.age) { return 1; }  
        else if (this.age < st.age) { return -1; }    }  
}
```

Provide data type of
compared object

Comparator<E>

- Allows you to provide **custom comparison logic**

```
class Dog implements Comparator<Dog>{  
    private String name;  
    private int age;  
  
    public int compare(Dog d, Dog d1) {  
        return d.age - d1.age;  
    }  
}
```


Problem: Comparable Book

- Expand Book by implementing **Comparable<Book>**
- Book have to be **compared by name**
 - When name is equal, **compare** them by **year**

```
<<Comparable<Book>>>  
Book
```

```
-title: String  
-year: int  
-authors: List<String>
```

```
+compareTo(Book): int
```

Solution: Comparable Book

```
public int compareTo(Book book) {  
    if (this.getTitle().compareTo(book.getTitle()) == 0) {  
        if (this.getYear() > book.getYear()) {  
            return 1;  
        } else if (this.getYear() < book.getYear()) {  
            return -1;  
        }  
    }  
    // Continues on the next slide
```

Solution: Comparable Book(2)

```
// ...  
return 0;  
    } else {  
        return this.getTitle().compareTo(book.getTitle());  
    }  
}
```

Problem: Book Comparator

- Create a class, which can **compare** two books
- Use your **BookComparator** to sort list of Books

```
<<Comparator<Book>>>  
BookComparator
```

```
+compare(Book, Book):int
```

Solution: Book Comparator

```
public class BookComparator implements Comparator<Book> {  
    @Override  
    public int compare(Book first, Book second) {  
        if (first.getTitle().compareTo(second.getTitle()) == 0) {  
            if (first.getYear() > second.getYear()) { return 1; }  
  
            // Continues on the next slide  
        }  
    }  
}
```


Solution: Book Comparator(2)

```
// ...  
else if (first.getYear() < second.getYear()) { return -1; }  
    return 0;  
} else {  
    return first.getTitle().compareTo(second.getTitle());  
}  
}  
}
```

Summary

- Variable arguments
- Iterable<T>
- Iterator<T>
- Comparable<T>
- Comparator<T>



Generics



Questions?



Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University Foundation
 - <http://softuni.foundation/>
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



**Software
University**



License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license



- Attribution: this work may contain portions from
 - "Fundamentals of Computer Programming with Java" book by Svetlin Nakov & Co. under CC-BY-SA license
 - "OOP" course by Telerik Academy under CC-BY-NC-SA license